Table Lookup for IP Addresses

Interaction with local applications

Mon, Jul 8, 2002

New system code uses a table lookup of IP addresses to eliminate the need for table searches and speed up IP support. Two local applications, however, are part of the picture, too. When the system is updated in a node, how can we know that the LAs are updated? If they are not, then the new scheme breaks down. This note develops a solution to potential problems of the IPNOD lookup table.

The `DSNQ` local application manages the IPNAT. When it receives a new IP address, it should update the new lookup table (IPNOD). Following system initialization, when all of the IPNAT is placed into the lookup table, this normally happens only when the DNS queries are made to refresh all the entries in the IPNAT. If a first-time node# is accessed, a new entry is added, `DNSQ` notices this, and it sends a query to the DNS. There seems no way to avoid adding a new call (to `SetNNode`) to `DNSQ`, since only that application can see the replies that come from the DNS.

The `AAUX` application must update the lookup table whenever a new IP address of an Acnet node# arrives. This normally occurs as a refresh of the trunk tables comes about via periodic (12 hours) queries of `OPER`. A new `AAUX` will need to call `SetANode` to update the lookup table for each IP address that is received. Again, there is no way that the system can notice this information's arrival and do this job, as only `AAUX` sees the replies from `OPER`.

It is desirable that applications not know about the lookup table in detail--where it is located, etc. But to make the calls to the Set routines named above, there must be new "jump table" entries in the system code to support such calls, or these routines must be added to `LASysLib`.

A local application, when it is initialized, can easily determine whether the underlying system code is a new enough version to have the features that the LA relies upon. In this case, the new versions of the LAs can easily insure that the system code is new enough to support the new jump table entries that give access to the two needed functions. If the system is not new enough, the application normally turns off its own enable bit, thus causing an immediate termination.

But what about vice versa? How can the underlying system code notice that the newer versions of the LAs are installed, since the system must rely upon the existence of the new versions to keep the lookup table up-to-date? The system code can know the version dates of these applications by consulting the CODES table entries. It can also find the LATBL entry to insure that it is enabled to be running, although we might just assume that it is. Another approach might be to use a message queue to communicate between the system and the LA; the new LA versions can create a message queue, and if the system sees that it exists, it can assume it will get the necessary updates so it can call the functions required.

One possibility is to use a message queue into which each LA could write from which the system could read. Suppose the new DNSQ creates a message queue into which it places advisories on new IP addresses it comes across. When the system wonders whether the new version of the LA exists, it can check for the existence of the message queue. If it exists, it can perform its new more efficient logic; if not, it can do things the old way. After the Data Access Table has been executed, there will need to be a check in the system code whether anything is in the message queue, so it can be processed accordingly. But it would be even better if this could be done sooner; *i.e.*, it would be better if the LA could make the call itself.

A routine such as `GetPNode` returns an error status in the case that it cannot use the lookup table to get the required info. This is true when the IP address upper 16 bits does not match the local Class B internet range. When `PSNIPARP` makes this call, and it finds that `GetPNode` cannot provide the info it needs, then it can be prepared to do things the old way, meaning it should do the search of the IPARP table. In most cases, this will not happen, so it doesn't matter much in terms of overall efficiency.

As for what error status `GetPNode` should return in place of the node number, in case the IP address is out of range for a quick lookup, we can use –1. (Zero would not work, since that is a valid return that means there is not a current IPARP entry for this IP address.)

A new idea for LA-to-system communication

Suppose the diagnostic data stream, used for recording unusual transitions of node number relationships with IP addresses, were also used for communications between the two LAs and the system. The LAs can determine whether the data stream exists and is enabled for writing. Instead of making the call to a Set routine, they instead place a new record into this same data stream. The format of the record would be special so that it would be recognizable by the system code as requesting service, rather than recording an unusual change in a node number. Code within the system would monitor this data stream, watching for such special entries. When one is seen, the appropriate Set call is made. This has an advantage over a message queue in that it can be easily monitored, because of the array of services that comprise data stream support.

Exactly what will these special records mean? Recall that the format of one of these records includes a new node number value and an old value. The special form could use an old value of `0xFF4E` or `0xFF41`. When this is seen, the system code makes a suitable call to either `SetNNode` or `SetANode`. For this purpose, the new value will never be zero. (Note that `0x4E`='N', `0x41`='A'.)

For the case of `AAUX`, there can be up to 256 IP addresses associated with each trunk table reply. But we would not want to deluge the data stream with so many records on a repeating basis, even if only every 12 hours, because this would compromise its diagnostic value. But `AAUX` can notice whether any IP address is new, while it is copying the new trunk table reply array of IP addresses into the system TRUNK table. Only if there is a change does it need to write into the data stream. In this way, the data stream will not be swamped with such unnecessary records.

The result is that the data stream becomes even more valuable, as it records the changes that the LAs detect. And it solves the problem of how to let the system know of news about new IP addresses relative to node numbers.

If a new system is run with old versions of the LAs, then no such data stream recording will occur. The system code can detect this by watching for a recent enough version date for the two LAs. But another way to do this is to again use the data stream. A new LA may, during initialization, write another special record into the data stream to advise the system code of its existence. The form of such a record could also use `0xFF4E` (for `SetNNode` signifying the new `DNSQ`) or `0xFF41` (for `SetANode` signifying the new `AAUX`) as the "old value" field. But the IP address would presumably be zero for this announcement record. This frees the system from having to analyze whether new versions of the LAs are running by checking version dates, etc. Armed with such knowledge, the system performs table searches the old way if necessary, but it uses the new faster method if the new LAs support it. Only with new LAs is the lookup table kept up-to-date. If the new version of either LA is disabled, its termination code can write a similar record using `0xF04E` or `0xF041`, say.

The place where system code can keep the record of the two new versions of the LAs for easy reference is in the upper bits of the new global variable word that holds the data stream index number in its low 12 bits. We can use the `0x8000` bit to signify the new `DNSQ` is open for business and use the `0x4000` bit to signify the new `AAUX` is operational.

The system code can include a call to a new routine in the new `IPNODGS` module that includes the package of Get and Set routines that deal with access to the lookup table. This routine is called `MonIPNOD`. Its job is to read from the data stream and process each record found there. To do this, it needs to remember where it last left off in its data stream monitoring. One place to remember this is in the data stream queue header, in the word after the `START` offset. This offset is a kind of `OUT` word, as used in many other system-based queues. Records would be read from the queue, using this `OUT` word as a reference, processed, advancing `OUT` until it is found to match `IN`. The records sought by this logic would only be the LA announcement records, resulting in setting or clearing the previous-mentioned flag bits, and the records placed there by `DNSQ` or `AAUX` that request a call to `SetNNode` or `SetANode`. Any other records found there would be diagnostic records only that can merely be ignored by `MonIPNOD`. The call to this monitoring routine is made from `Update` before starting to process the Data Access Table. If the data stream queue is there, but the saved offset is zero, it means that it has never been initialized yet, so it should be set at that time to match `START` before beginning to monitor any records.

What is a sensible order for the 3 key fields of a data stream record?
    IP address
    new value
    old value

Think of this order as corresponding to a call to a Set routine, in which the first argument is the IP address, the second is the new value, and the value returned is the old value.

The "open for business" announcement records would look like
    `0000 0000 0000 FF4E`, for the `DNSQ` case, or
    `0000 0000 0000 FF41`, for the `AAUX` case.

The "closing" announcements would look like:
    `0000 0000 0000 F04E` (`DNSQ`) or
    `0000 0000 0000 F041` (`AAUX`)

The advisory records would look like
    `83E1 xxxx nnnn FF4E` (`DNSQ`) or
    `83E1 xxxx nnnn FF41` (`AAUX`)

Here the `83E1xxxx` is the IP address, and nnnn is the node number. (Of course the `83E1` would be a different value for system operation outside of Fermilab.) The `4E`='`N`' implies that `SetNNode` should be called; the `41`='`A`' implies that `SetANode` should be called.

The diagnostic records of unusual node number changes would look like
    `83E1 xxxx nnnn yyyy`

If nnnn is in the `0500–07FF` range, it was written by `SetNNode`.
If nnnn is in the `0900–10FF` range, it was written by `SetANode`.
If nnnn is in the `6000–6FF0` range, it was written by `SetPNode`.
Here, `yyyy` represents a nonzero "old value" that was found different from the nonzero "new value" during a Set routine call.

The advantage to this scheme is that different versions of the system should work correctly with different versions of the LAs, during the transition period, with the only difference being one of efficiency. As new systems are installed, and new applications are also installed, the table lookup method of saving searches on IP addresses should improve efficiency in all nodes, especially in the PowerPC nodes that have relatively slow access (1 μs) to the IPNAT, TRUNK, and IPARP tables that reside in nonvolatile memory.

Performance measurements

The motivation for making the changes described here is to improve the efficiency of system operation. But there is no such measurement scheme in place; what's more, it is less than perfectly clear how often the various search routines are called. The code must be "instrumented" in order to shed some light on this.

The search routines that relate to the IPNOD and IPADD tables are four, as mentioned above. We need to measure the execution time of each of these routines and count how often they are called. Here is the scheme that has been implemented:

The data stream queue has a user area that can be made large enough to hold some diagnostic timing records. Each of the four routines measures its own elapsed time each time it is called. To facilitate this, a new routine called `MicroSec` is used, a function with no arguments that returns a 32-bit microsecond counter obtained from the timer on the MVME-162 board that is commonly used for this purpose. Each of the four routines calls this function at the beginning and at the end to get an elapsed time. (The elapsed time difference will of course fit within 16 bits.) A routine `IPTLOG` has been written to facilitate the reporting of these results.

```
PROCEDURE IPTLOG(index: Integer; usec: Integer; node: Integer);
```

The index argument is a small number in the range 1–4 that maps to the four routines:

1	`PsnIPARP`
2	`NodeIPN`
3	`GtNodeN`
4	`FindUDP`

The user area of the data stream queue header is laid out in 8-byte structures as follows:

The first 4 bytes show the elapsed time for the `InzIPNOD` routine that is called only once at system initialization. The memory used by the tables is large, so this may not be a small number. The next four bytes have no use at this time. After these 8 bytes, there follows space for four more 8-byte records, one for each of the 4 routines in the above order. The minimum user area needed in the data stream queue header is therefore 40 bytes.

Field	*Size*	*Meaning*
`nodNum`	2	Node# relating to the elapsed time measurement
`countr`	2	Counter incremented every time an entry is updated
`mxTime`	2	Maximum elapsed time encountered, in μs
`lsTime`	2	Last elapsed time recorded, in μs

As always, one must realize that an maximum time measurement can be influenced by the time spent is an interrupt routine, or of a higher priority task.

Preliminary results are as follows, with the code installed in a 68040-based IRM.

Typical times for `PsnIPARP` executions are 20–30 μs. Any UDP datagram that is received must have a pseudo node# determined from its source IP address and source UDP port#. `PsnIPARP` does this. If the flag bit is turned off, which prevents `PsnIPARP` from calling `GetPNode`, the times might be more like 30–40 μs. It depends somewhat on how deeply the matching entry is found in IPARP table. This is not a serious difference, but the IRM's nonvolatile memory is not so slow, either.

For one example, the `IPNodeN` timing was 48 μs the "fast" way but 82 μs the "slow" way. Recall that this routine is needed when a request is initiated that targets a native node#, which happens much less often than the first one.

The `GtNodeN` timing example showed 10 μs the fast way and 42 μs the slow way. This has to be done by `Classic` or `ACReq` to get the native node# for an incoming datagram given its source IP address. When receiving replies from other nodes, one will see both the `PsnIPARP` entry and this one counting actively. For a node like the Linac server `node0600`, these entries can be extremely active.

The `FindUDP` timing showed an example time of 11 μs the fast way and 110 μs the slow way. This one is needed when initiating an Acnet request. The times presumably depend upon where in the TRUNK tables the sought-for IP address is located.

The elapsed time for `InzIPNOD` was about 33 milliseconds. The two tables occupy 640K bytes.

These preliminary measurements don't show a large problem for IRMs. But we have been hurt before by slow access to nonvolatile memory in the MVME-2401 boards that have a PMC nonvolatile memory board installed. But even for an IRM, knowing that searches are not relied upon for network support somehow seems better.